

# C-Syntax I

## Statements & Control Flow

# Table of Contents

Recap

Data Types & Declarations

Operands & Operators: Expressions

Statements & Blocks, Scoping

Control Flow: If-Else, Loops

Functions: Declaration, Definition

Representation of Data & Formatted Output

# Recap

Wir hatten folgendes Programm:

```
1 int main(void)           # Funktionsdef. von `main`
2 {
3     puts("Hello World"); # Funktionsaufruf von `puts`
4     return (0);         # Beendet `main`: "Ergebnis" 0
5 }
```

## Ausblick

Wir beschäftigen uns nun damit wie wir ...

- ▶ ... mit den *Datentypen* neue Variablen *deklarieren* können
- ▶ ... mit diesen nun wieder *Ausdrücke* berechnen können
- ▶ ... damit dann weitere *Anweisungen* an den Computer geben können
- ▶ ... diese wiederholt ausführen oder nur unter Bedingungen, indem wir *Kontrollanweisungen* einführen
- ▶ ... komplett neue *Funktionen definieren* können

# Data Types

- ▶ Der Typ einer Variable gibt an *was* dieser *wie* speichert.
- ▶ Grundlegend gibt es in C eigentlich nur Zahlen
- ▶ Dies spiegelt den Aufbau eines Computers wieder
- ▶ Buchstaben sind bspw. nur Zahlen, von denen wir wissen, dass wir sie als Buchstaben zu interpretieren haben!
- ▶ Das nennt man: „Sie sind codiert“, s. ASCII-Code
- ▶ Alle anderen Datentypen können aus den primitiven zusammengesetzt werden

## Data Types: Beispiel

- ▶ Explizit kennen wir schon den Datentypen `int`, für den Rückgabewert von `main()`
- ▶ Ein Integer ist eine Ganzzahl, kann also negativ, positiv oder 0 sein
- ▶ Die Bedeutung des Rückgabewerts von `main()` ist speziell für C-Programme: Exit-Code des gesamten Programms
- ▶ Es gibt noch andere, weniger offensichtliche Typen in diesem Beispiel
- ▶ Wir beschäftigen uns aber erstmal mit den grundlegenden primitiven Typen

# Primitive Data Types

Einfachste Datentypen:

- ▶ `char`: Ein Byte, häufig für ASCII-Buchstaben verwendet
- ▶ `_Bool`: Boolean, kann nur 0 oder 1 halten, trotzdem so groß wie `char`
- ▶ `int`: Ganzzahl, mit negativen Zahlen
- ▶ `float`: Gleitkommazahl einfacher Präzision
- ▶ `double`: Gleitkommazahl doppelter Präzision

Modifikatoren für `int`:

- ▶ mit `unsigned` speichert der Typ nur natürliche Zahlen
- ▶ mit `short`, `long` und `long long` kann die sog. Größe des Typs verändert werden

## Größe eines Typs

- ▶ Üblicherweise sind ein Byte acht Bit, es können also maximal  $2^8$  unterschiedliche Zahlen repräsentiert werden
- ▶ Brauchen wir mehr, brauchen wir ggf. `int`
- ▶ Größe ist plattformabhängig! Kleine Microchips haben ggf. andere Standardgrößen für Ganzzahlen als „normale“ Rechner
- ▶ Es gibt auch Typen mit festen Größen, je nach Anwendung bietet sich das an.
- ▶ Diese fallen unter die Kategorie der abgeleiteten Datentypen

## Abgeleitete Datentypen

Aus `<stddef.h>`

- ▶ `intN_t` hat genau  $N$  Bits, wobei  $N \in \{8, 16, 32, 64\}$ .
- ▶ `uintN_t` analog, vorzeichenlos
- ▶ `[u]intmax_t` größter verfügbarer Datentyp für Ganzzahlen

Aus `<stdint.h>`

- ▶ `size_t` Ebenfalls vorzeichenlos, groß genug um die Größe jedes Objektes als Zahl zu speichern
- ▶ Es ist häufig sinnvoll also zur Programmlaufzeit selber diese Größe zu wissen!

# Deklarationen

- ▶ Um mit Variablen Arithmetik zu betreiben, müssen wir sie formal deklarieren
- ▶ Dies gibt zur *Compilezeit* die benutzten Variablen bekannt
- ▶ Deklarationen werden mit Semikoli terminiert
- ▶ Einfache Deklarationen beinhalten nur den Typen der Variable, gefolgt von dem Variablennamen

## Operands & Operators: Expressions

- ▶ Nun da wir (arithmetische) *Operanden* haben, benötigen wir noch *Operatoren*
- ▶ Zusammen haben wir (arithmetische) *Ausdrücke*
- ▶ Operanden können
  - ▶ Konstanten (42, '?' ) und Stringlitterale ("FOO"),
  - ▶ Variablen, oder auch
  - ▶ weitere Ausdrücke sein.
- ▶ Operatoren sind nicht nur arithmetische Operatoren, sondern auch Zuweisungen und Abfragen oder Konditionale.

## Expression Evaluation

- ▶ Die Auswertung einer Expression berechnet ihren Wert
- ▶ Hierbei werden die einzelnen Teile der Expression wiederum einzeln ausgewertet
- ▶ ...also auch Funktionsausführungen, Zuweisungen, etc.

```
1 int    a = 42;
2 int    b =  8;
3 int    c = (a+b)/5 + (7*3) + 12*5/3 + atoi("24");
4 _Bool  d = a == c;
5 int    e = (a = b);
6 b = c;
7 a == c;
8 42;
```

## Statements

- ▶ Ein Ausdruck an sich kann nie ausgeführt werden
- ▶ Hierzu benötigen wir eine Anweisung
- ▶ Einfachste Anweisungen sind Ausdrücke mit einem Semikolon beendet
- ▶ Jedes Statement wird sequentiell abgearbeitet

```
1 int    a = 42;  
2 int    b =  8;  
3 int    c = (a+b)/5 + (7*3) + 12*5/3 + atoi("24");  
4 _Bool  d = a == c;  
5 int    e = (a = b);  
6 b = c;  
7 a == c;  
8 42;
```

## Compound Statements: Blocks

- ▶ Statements werden in Blöcken gruppiert
- ▶ Ein Block ist äquivalent zu einem Statement und kann anstelle eines Statements auftreten
- ▶ Blöcke werden mit geschweiften Klammern umschlossen
- ▶ Haben Eigenschaften bezüglich der „Sichtbarkeit“ von Variablen

```
1 int a = 42;  
2 int d;  
3 {  
4     int b = 10;  
5     d = b/2 + 42;  
6 }
```

# Scoping

- ▶ Scope: Variablen sind nur sichtbar innerhalb ihres Blockes!
- ▶ Nur darin kann man sich auf die Variable über ihren Namen beziehen.
- ▶ Weitere Vernerstung von Unterblöcken ist aber okay!

```
1 int a = 42;
2 int d;
3 {
4     int b = 10;
5     d = b/2 + 42;
6 }
7 int e = a + d; // b und c nicht sichtbar!
```

# Control Flow

- ▶ Wenn wir von „Kontrollfluss“ reden, meinen wir die Abfolge von „Statements“ bzw. Blöcken
- ▶ Um Abfolge der Statements zu verändern gibt es Control Statements
- ▶ Das sind z. Bsp. If-Else und Schleifen (For, While)
- ▶ Ein If-Statement kontrolliert, ob das folgende Statement ausgeführt werden soll
- ▶ Hierfür evaluiert es eine Expression auf Ungleichheit mit 0 (true)
- ▶ Analog wiederholt eine While-Schleife das Statement, solange die Bedingung wahr ist
- ▶ Äquivalent können wir anstatt der Statements auch Blöcke kontrollieren
- ▶ ... welche wiederum aus anderen (Control-) Statements bestehen

```
1 int a = 0;
2 if (a == 0)
3     puts("a = 0");
4 while (a < 10)
5     a = a + 1;           // alt.: a += 1, ++a / a++
6 for (int i = 0; i < 10; i++)
7     puts("i < 10");
8     puts("foo");       // Nicht mehr i.d. Schleife!
9
10 while (a > 0) {        // Block anstatt Statement
11     puts("a > 0");
12     a--;
```

# Functions

- ▶ Eine weitere Unterteilung des Programmes bieten Funktionen
- ▶ Diese berechnen anhand ihrer als Argumente übergebenen Parameter ihren Rückgabewert
- ▶ viele Funktionen in C haben aber auch Nebenwirkungen
- ▶ z. Bsp. `puts()`:
  - ▶ probiert den übergebenen String auf der Konsole auszugeben
  - ▶ bei Fehler gibt es die Konstante EOF zurück und setzt div. Fehlervariablen
  - ▶ sonst gibt es eine nicht-negative Zahl aus.
- ▶ Die Argumente des Funktionsaufrufes müssen in der Reihenfolge und vom Typen mit der Spezifikation übereinstimmen

## Function Definition

- ▶ Häufig haben Funktionsdefinitionen die Form:  
*Typ Funktionsname(Typ Name, ...) Block*
- ▶ Bspw.: `int leapyear(int year) /* ... */`
- ▶ Wenn die Funktion keine Argumente nehmen soll, hat sie den speziellen Parameter `void`!
- ▶ Das Gleiche gilt für den Rückgabewert, sollte die Funktion nichts „berechnen“
- ▶ Falls es jedoch einen gibt, kann der in der Funktion mit dem Statement `return (/* 42? */);` zurückgegeben werden.
- ▶ Ansonsten einfach `return` zum vorzeitigen Beenden der Funktion

## Function Declaration

- ▶ Häufig haben Funktionsdefinitionen die Form:  
*Typ Funktionsname(Typ Name, ...) Block*
- ▶ Allgemeiner jedoch, kann der sog. *Deklarator* (zwischen Typ und Block) deutlich komplexer sein!
- ▶ Bewusst abschreckendes Beispiel:

```
1 void (*signal(int sig, void (*func)(int)))(int);
```

- ▶ Ich habe jetzt an Stelle eines Blockes ein Semikolon am Ende gesetzt
- ▶ ... das ist damit nur eine Deklaration.

## How the Compiler Works: Function Pre-Declaration

- ▶ C-Compiler beginnt am Anfang eurer C-Datei und liest ein
- ▶ Für jede neue Deklaration: Hinterlegt Namen des Objekts in einer Tabelle
- ▶ Wird Objekt später benutzt, kann der Compiler überprüfen, ob die Typen stimmen (was soll eine Division einer Funktion durch einen String sein?)
- ▶ Wird das Objekt aber benutzt *bevor* es bekannt ist, kann der Compiler nicht arbeiten.
- ▶ Am Rande: Funktionsdefinitionen und -deklarationen können nicht in Blöcke geschrieben werden
- ▶ Sind also erstmal „global“ sichtbar!

```
1 int main(void)
2 {
3     foo(42, 24); // Unbekannte Funktion foo!
4     /* ... */
5 }
6
7 int foo(int bar, int baz) // Definition mit Deklaration
8 {
9     return (bar+baz);
10 }
```

```
1 int foo(int bar, int baz) // Definition mit Deklaration
2 {
3     return (bar+baz);
4 }
5
6 int main(void)
7 {
8     foo(42, 24); // Bereits bekannte Funktion foo
9     /* ... */
10 }
```

```
1 int foo(int bar, int baz); // Deklaration
2
3 int main(void)
4 {
5     foo(42, 24); // Bereits bekannte Funktion foo
6     /* ... */
7 }
8
9 int foo(int bar, int baz) // Definition mit Deklaration
10 {
11     return (bar+baz);
12 }
```

## Representation of Data

Was bedeutet es eigentlich, wenn die Zahl 42 in einem `unsigned char` gespeichert ist?

- ▶ Die Größe des Typs gibt an, wie viel Speicher (in Bytes) im Rechner dafür benutzt wird
- ▶ Die Zahl 42 wird „klassisch“ schlicht in der Basis 2 encodiert: 01000010
- ▶ Dies könnte mit anderen Interpretationen eine andere Bedeutung haben als 42
- ▶ Bspw. wäre das zugehörige ASCII-Zeichen `'*'`.
- ▶ Jede andere Repräsentation ist möglich: Denkt euch selbst was aus

## Common Formats

- ▶ Häufige Interpretationen sind hexadezimale, dezimale und oktale Ganzzahlen (wie wird das Minus gespeichert?), Fließkommazahlen oder ganze Strings
- ▶ Wie die genau funktionieren, gucken wir uns bei Strings später genauer an (Problem: Die können beliebig lang sein!)
- ▶ Für alle diese Interpretationen gibt es jedoch schon vorgefertigte Funktionen die Umwandeln
- ▶ Um die Bytes „schön“ auszugeben gibt es die universale Funktion `printf()`.
- ▶ Für das Gegenteil gibt es `scanf()`

## Formatted Output

- ▶ `printf()` nimmt als erstes Argument einen sog. Formatstring, welcher aus normalen Zeichen und Platzhaltern besteht.
- ▶ Für jeden Platzhalter wird ein weiteres Argument benötigt, das entsprechend des Platzhalters interpretiert wird.
- ▶ `printf("Zahl %d als ASCII-Zeichen: '%c'\n", 42, 42)`
- ▶ Gibt aus: „Zahl 42 als ASCII-Zeichen: '\*'“
- ▶ Platzhalter sind `"%d"` und `"%c"`
- ▶ Geben den gleichen Wert als Dezimalzahl und als ASCII-Zeichen interpretiert aus
- ▶ Das `"\n"` beendet die Zeile mit einem Zeilenumbruch